C++学习之动态数组类的封装

2014-07-28 | Lang.-Cpp |

动态数组（Dynamic Array）是指动态分配的、可以根据需求动态增长占用内存的数组。为了实现一个动态数组类的封装，我们需要考虑几个问题：**new/delete的使用、内存分配策略、类的四大函数（构造函数、拷贝构造函数、拷贝赋值运算符、析构函数）、运算符的重载**。涉及到的知识点很多，鉴于本人水平有限，在这里只做简单的介绍。

内存分配策略

当用new为一个动态数组申请一块内存时，数组中的元素是**连续存储**的，例如 vector和string。当向一个动态数组添加元素时，如果没有空间容纳新元素，不可能简单地将新元素添加到内存中的其他位置——因为元素必须连续存储。所以**必须重新分配一块更大的内存空间，将原来的元素从旧位置移动到新空间中，然后添加新元素，释放旧的内存空间**。如果我们每添加一个新元素，就执行一次这样的内存分配和释放操作，效率将会慢到不行。

为了避免上述的代价，必须减少内存重新分配的次数。所以我们采取的策略是：**在不得不分配新的内存空间时，分配比新的空间需求更大的内存空间（通常为2倍）**。这样，在相当一段时间内，添加元素时就不用重新申请内存空间。注意，只有当迫不得已时才可以分配新的内存空间。

类的四大函数

一个C++类一般至少有四大函数，即构造函数、拷贝构造函数、拷贝赋值运算符、析构函数。如果类未自己定义上述函数，C++编译器将为其合成4个默认的版本。但是往往编译器合成的并不是我们所期望的，为此我们有必要自己定义它们。

构造函数

```
class Foo {
public:
        Foo();  // 构造函数
        Foo(string &s);  // 接受一个参数的构造函数
        // ...
};
```

构造函数的名字和类名相同，没有返回类型。类可以包含多个构造函数（重载），它们之间在参数数量或类型上需要有所区别。**构造函数有一个初始化部分和一个函数体**，成员的初始化是在函数体执行之前完成的。

拷贝构造函数

如果一个构造函数的第一个参数是**自身类类型的引用**，且任何额外参数都有默认值，则此构造函数是**拷贝构造函数（copy constructor）**。

```
class Foo {
public:
        Foo();
        Foo(const Foo&);  // 拷贝构造函数
        // ...
};
```

拷贝构造函数定义了如何用一个对象**初始化**另一个同类型的对象。拷贝初始化通常使用拷贝构造函数来完成。**拷贝初始化**发生在下列情况中：

- 使用等号（=）初始化一个变量
- 将一个对象作为实参传递给一个非引用类型的形参
- 从一个返回类型为非引用类型的函数返回一个对象
- 用花括号列表初始化一个数组中的元素

拷贝赋值运算符

类的**拷贝赋值运算符（copy-assignment operator）**是一个名为 `operator=` 的函数。类似于其他任何函数，它也有一个返回类型和一个参数列表。

```
class Foo {
public:
        Foo();
        Foo& operator=(const Foo&);  // 赋值运算符
```

```
5           // ...
6   };
```

拷贝赋值运算符定义了如何将一个对象**赋值**给另一个同类型的对象。赋值运算符是一个成员函数也是一个二元运算符，其左侧运算对象就绑定到隐式的this指针，右侧运算对象作为显式参数传递。**注意**：为了与内置类型的赋值保持一致，赋值运算符通常返回一个指向其左侧运算对象的引用。

析构函数

类的**析构函数（destructor）**用来释放类对象使用的资源并销毁类对象的非static数据成员，无论何时只要一个对象被销毁，就会自动执行析构函数。

```
1   class Foo {
2   public:
3           ~Foo();  // 析构函数
4           // ...
5   };
```

析构函数的名字由波浪号（~）加类名构成，也没有返回类型。由于析构函数不接受参数，因此它不能被重载。**析构函数有一个函数体和一个析构部分**，销毁一个对象时，首先执行析构函数体，然后按初始化顺序的逆序销毁成员。

运算符的重载

重载的运算符是具有特殊名字的函数：**它们的名字由关键字 operator 和其后要定义的运算符号共同组成**。和其他函数一样，重载的运算符也包含返回类型、参数列表、函数体，比如拷贝赋值运算符。

当我们定义重载的运算符时，必须首先决定是将其声明为类的成员函数还是声明为一个普通的非成员函数。有些运算符必须作为成员，而另一些运算符作为普通函数比作为成员更好：

- 赋值（=）、下标（[ ]）、调用（( )）和成员访问箭头（->）运算符必须是成员。
- 复合赋值运算符一般来说应该是成员，但并非必须，这一点与赋值运算符略有不同。
- 改变对象状态的运算符或者与给定类型密切相关的运算符，如递增、递减、解引用运算符，通常应该是成员。
- 具有**对称性**的运算符可能转换任意一端的运算对象，例如算术、相等性、关系和位运算符等，因此它们通常应该是普通的非成员函数。

当然，除了赋值运算符之外，我们还需要为动态数组定义**下标运算符operator []**。下标运算符必须是成员函数。为

了让下标可以出现在赋值运算符的任意一端，下标运算符函数通常返回所访问元素的引用。

动态数组类的封装

下面给出了动态数组DArray类的接口：

```
class DArray
{
private:
        double *m_Data;  // 存放数组的动态内存指针
        int m_Size;      // 数组的元素个数
        int m_Max;       // 预留给动态数组的内存大小
private:
        void Init();     // 初始化
        void Free();     // 释放动态内存
        inline bool InvalidateIndex(int nIndex);  // 判断下标的合法性
public:
        DArray();        // 默认构造函数
        DArray(int nSize, double dValue = 0);  // 构造函数，设置数组大小，默认值为dValue
        DArray(const DArray& arr);  // 拷贝构造函数
        DArray& operator=(const DArray& arr);  // 拷贝赋值运算符
        ~DArray();       // 析构函数

        void Print();    // 输出显式所有数组元素的值
        int GetSize();   // 获取数组的大小（元素个数）
        void SetSize(int nSize);  // 重新设置数组的大小
        double GetAt(int nIndex);  // 获取指定位置元素
        void SetAt(int nIndex,double dValue); // 重置指定元素的值
        void PushBack(double dValue);  // 追加一个新元素到数组末尾
        void DeleteAt(int nIndex);     // 删除指定位置地元素
        void InsertAt(int nIndex, double dValue);  // 插入一个新的元素到数组中
        double operator[](int nIndex) const;      // 重载下标运算符[]
};
```

下面是我的实现：

```
 void DArray::Init()
 {
         m_Size = 0;   // 默认情况下数组不包含元素
         m_Max = 1;
         m_Data = new double[m_Max];
 }

 void DArray::Free()
```

```
 9  {
10          delete [] m_Data;
11  }
12
13  bool DArray::InvalidateIndex(int nIndex)
14  {
15          if(nIndex>=0 && nIndex<m_Size)
16                  return false;
17          else
18                  return true;
19  }
20
21  // 默认构造函数
22  DArray::DArray()
23  {
24          Init();
25  }
26
27  // 构造函数
28  DArray::DArray(int nSize, double dValue)
29  {
30          if(nSize == 0)
31                  Init();
32          else
33          {
34                  m_Size = nSize;
35                  m_Max = nSize*2;
36                  m_Data = new double[m_Max];
37                  for(int i=0; i<nSize; ++i)
38                          m_Data[i]=dValue;
39          }
40  }
41
42  // 拷贝构造函数
43  DArray::DArray(const DArray& arr)
44  {
45          m_Size = arr.m_Size;  /*复制常规成员*/
46          m_Max = arr.m_Max;
47          m_Data = new double[m_Max];   /*复制指针指向的内容*/
48          memcpy(m_Data, arr.m_Data, m_Size*sizeof(double));
49  }
50
51  // 拷贝赋值运算符
52  DArray& DArray::operator=(const DArray& arr)
53  {
54          if(this == &arr)  /*自赋值*/
55                  return *this;
56          m_Size = arr.m_Size;
57          m_Max = arr.m_Max;
```

```
58          /* 先将右侧对象拷贝到临时对象中，然后再销毁左侧对象*/
59          double *m_Temp = new double[m_Max];
60          memcpy(m_Temp, arr.m_Data, m_Size*sizeof(double));
61          delete [] m_Data;
62          m_Data = m_Temp;
63
64          return *this;
65  }
66
67  // 析构函数
68  DArray::~DArray()
69  {
70          Free();
71  }
72
73  // 打印数组
74  void DArray::Print()
75  {
76          if(m_Size == 0)
77          {
78                  cout << "Error: The empty array can't be Printed." << endl;
79                  exit(0);
80          }
81          else
82          {
83                  for(int i=0; i<m_Size; ++i)
84                          cout << m_Data[i] << " ";
85                  cout << endl;
86          }
87  }
88
89  // 获取数组大小
90  int DArray::GetSize()
91  {
92          return m_Size;
93  }
94
95  // 重置数组大小
96  void DArray::SetSize(int nSize)
97  {
98          if(nSize < m_Size)   /*截断*/
99          {
100                 for(int i=nSize; i<m_Size; ++i)
101                         m_Data[i] = 0;
102         }
103         if(m_Size<=nSize && nSize<=m_Max)  /*新增元素置0*/
104         {
105                 for(int i=m_Size; i<nSize; ++i)
106                         m_Data[i] = 0;
```

```
107         }
108         if(nSize > m_Max)   /*需要重新分配空间*/
109         {
110                 m_Max = nSize*2;
111                 double *temp = new double[m_Max];
112                 memcpy(temp, m_Data, m_Size*sizeof(double));
113                 for(int i=m_Size; i<nSize; ++i)
114                         temp[i] = 0;
115                 delete [] m_Data;
116                 m_Data = temp;
117         }
118         m_Size = nSize; /*设置数组大小*/
119 }
120
121 // 获取指定位置元素
122 double DArray::GetAt(int nIndex)
123 {
124         if(InvalidateIndex(nIndex))
125         {
126                 cout << "Error: the index of GetAt is invalid!" << endl;
127                 exit(0);
128         }
129         return m_Data[nIndex];
130 }
131
132 // 设置指定位置元素的值
133 void DArray::SetAt(int nIndex, double dValue)
134 {
135         if(InvalidateIndex(nIndex))
136         {
137                 cout << "Error: the index of SetAt is invalid!" << endl;
138                 exit(0);
139         }
140         else
141         {
142                 m_Data[nIndex] = dValue;
143         }
144 }
145
146 // 追加一个新元素到数组末尾
147 void DArray::PushBack(double dValue)
148 {
149         if(m_Size < m_Max)
150         {
151                 m_Data[m_Size] = dValue;
152         }
153         else
154         {
155                 m_Max = m_Max*2;
```

```
156                 double* temp = new double[m_Max];
157                 memcpy(temp, m_Data, m_Size*sizeof(double));
158                 delete [] m_Data;
159                 m_Data = temp;
160                 m_Data[m_Size] = dValue;
161         }
162         ++m_Size;  /*数组大小加1*/
163 }
164
165 // 从数组中删除一个元素
166 void DArray::DeleteAt(int nIndex)
167 {
168         if(InvalidateIndex(nIndex))
169         {
170                 cout << "Error: the index of DeleteAt is invalid." << endl;
171                 exit(0);
172         }
173         else
174         {
175                 for(int i=nIndex; i<m_Size; ++i)
176                         m_Data[i] = m_Data[i+1];
177                 m_Data[m_Size-1] = 0;
178                 --m_Size;
179         }
180 }
181
182 // 插入一个新元素到指定位置
183 void DArray::InsertAt(int nIndex, double dValue)
184 {
185         if(nIndex<0 || nIndex>m_Size)
186         {
187                 cout << "Error: the index of InsertAt is invalid!" << endl;
188                 exit(0);
189         }
190
191         if(m_Size < m_Max)  /* 未满，插入 */
192         {
193                 for(int i=m_Size-1; i>=nIndex; --i)
194                         m_Data[i+1] = m_Data[i];
195                 m_Data[nIndex] = dValue;
196         }
197         else                /* 重新分配空间 */
198         {
199                 m_Max = m_Max*2;
200                 double* temp = new double[m_Max];
201                 memcpy(temp, m_Data, m_Size*sizeof(double));
202                 delete [] m_Data;
203                 m_Data = temp;
204                 for(int i=m_Size-1; i>=nIndex; --i)
```

```
205                         m_Data[i+1] = m_Data[i];
206                 m_Data[nIndex] = dValue;
207         }
208         ++m_Size;  /* 数组大小加1 */
209 }
210
211 // 重载下标运算符[]
212 double DArray::operator[](int nIndex) const
213 {
214         if(nIndex<0 || nIndex>=m_Size)
215         {
216                 cout << "Error: the index in [] is invalid!" << endl;
217                 exit(0);
218         }
219         return m_Data[nIndex];
220 }
```

经过简单的测试，暂时还没有发现Bug。可能测试并不全面，如果你发现了问题，希望你能在评论里告诉我，万分感谢！！！

附：String类的实现

C++ 的一个常见面试题是让你实现一个 String 类，限于时间，不可能要求具备 std::string 的功能，但至少要求能正确管理资源。

如果你弄懂了上面DArray类的写法，那么实现String类应该就不难了。因为面试官一般只是想考查你能不能正确地写出构造函数、析构函数、拷贝构造函数、拷贝赋值运算符以及+、[]、<<、>>运算符重载等等。下面给出一个String类的接口，你可以自己试试手实现一下：

```
1  class String{
2      friend ostream& operator<< (ostream&,String&); //重载<<运算符
3      friend istream& operator>> (istream&,String&); //重载>>运算符
4  public:
5      String();   // 默认构造函数
6      String(const char* str);       // 带参构造函数
7      String(const String& rhs);        // 拷贝构造函数
8      String& operator=(const String& rhs);   // 拷贝赋值运算符
9      String operator+(const String& rhs) const;  //operator+
10     bool operator==(const String&);              //operator==
11     bool operator!=(const String&);              //operator!=
```

```
12      char& operator[](unsigned int);              //operator[]
13      size_t size() const;
14      const char* c_str() const;
15      ~String();    // 析构函数
16  private:
17      char *m_data; // 用于保存字符串
18  };
```

下面是String类的实现，注意一些得分点：

```
1   // 默认构造函数
2   String::String()
3           :m_data(new char[1])
4   {
5           m_data[0] = '\0';  /*得分点：空字符串存放结束标志'\0'*/
6   }
7
8   // 带参的构造函数
9   String::String(const char* str)
10          :m_data(new char[strlen(str)+1])
11  {
12          strcpy(m_data,str);
13  }
14
15  // 拷贝构造函数
16  String::String(const String& rhs)
17  {
18          m_data = new char[strlen(rhs.m_data)+1];
19          strcpy(m_data, rhs.m_data);
20  }
21
22  // 拷贝赋值运算符
23  String& String::operator=(const String& rhs)
24  {
25          if(this == &rhs)  /*得分点：自赋值*/
26                  return *this;
27          delete [] m_data; /*得分点：释放左侧对象资源*/
28          m_data = new char[strlen(rhs.m_data)+1];
29          strcpy(m_data, rhs.m_data);
30          return *this;
31  }
32
33  // 析构函数
34  String::~String()
35  {
36          delete [] m_data;
37  }
```

```
38
39  // operator+
40  String String::operator+(const String& rhs) const
41  {
42          String newStr;
43          newStr.m_data = new char[strlen(m_data)+strlen(rhs.m_data)+1];
44          strcpy(newStr.m_data, m_data);
45          strcat(newStr.m_data, rhs.m_data);
46          return newStr;
47  }
48
49  // operator==
50  bool String::operator==(const String& rhs)
51  {
52          if(strcmp(m_data, rhs.m_data) == 0) /*相等返回0*/
53                  return true;
54          else
55                  return false;
56  }
57
58  // operator!=
59  bool String::operator!=(const String& rhs)
60  {
61          return !(*this == rhs);
62  }
63
64  // operator[]
65  char& String::operator[](unsigned int index)
66  {
67          if(index>=0 && index<strlen(m_data))
68          {
69                  return m_data[index];
70          }
71          else
72          {
73                  cout << "Error: the index of [] is invalid." << endl;
74                  exit(0);
75          }
76  }
77
78  // 获取String大小
79  size_t String::size() const
80  {
81          return strlen(m_data);
82  }
83
84  // 获取C风格字符串
85  const char* String::c_str() const
86  {
```

```
87          return m_data;
88  }
89
90  // 重载<<运算符
91  ostream& operator<<(ostream& os, String& str)
92  {
93          os << str.m_data;
94          return os;
95  }
96
97  // 重载>>运算符
98  istream& operator>>(istream& is, String& str)
99  {
100         char temp[255];
101         is >> temp;
102         str = temp;
103         return is;
104 }
```

#Cpp

C++学习之new与delete、malloc与free

C++学习之普通函数指针与成员函数指针